STATISTICAL MODEL FOR ESTIMATION OF
HALSTEAD'S SOFTWARE SCIENCE LENGTH

by

JOJO THOPPIL KUNJUPALU

B.E., University of Madras, 1983
M.S., Kansas State University, 1986

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:

Major Professor

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

Contents   Page Number

INTRODUCTION

SOFTWARE ENGINEERING

Software engineering is a subject that has been cloaked in mystery since the introduction of the term in the late 1960's. Papers and books have been written and conferences have been presented extolling the virtues of software engineering as a panacea for the problems that have been associated with software development over the last two decades. Several definitions have been proposed for software engineering. Though no two definitions are identical, they all seem to be linked with the fact that software engineering involves practical application of techniques, in a very engineering-like fashion. Taking into consideration the relation to engineering, F. L. Bauer of the Technical University, Munich, Germany, defines software engineering as

"The establishment and use of sound engineering
principles (methods) in order to obtain economically
software that is reliable and works on real machines."
[Bau72]

SOFTWARE MEASURES

Software science is a field of software engineering. Software science is related to information theory. The objective of software science is to place quantitative measures on basic or intrinsic properties of software. This quantitative measure, better known as "software metrics", is calculated from the specification, design, code, or documentation of a computer program. Software measures deal with quantifying various aspects of computer software and its development. As defined by Boehm

"A software metric is a measurement of software, ie., a measure of the extent or degree to which a product possesses and exhibits a certain quality, property, or attribute." [Boe78]

Software measures give a quantitative view of software and its development. They can be used to improve and refine software methods and tools. However, as stated by Michael L. Cook, "the metrics currently available should only be used as guides to software development and maintenance. They should not be used as rigid, unquestionable measures that replace human judgement."

Software measures can be broadly categorized into

1) numbers that predict

    eg., predicting system change

          predicting program complexity

          predicting programming effort;

2) numbers related to human understanding

    eg., program correctness

          program testability

          program maintainability

          program flexibility

          program accuracy

3) numbers that help in management

    eg., resource estimation

          cost of development

          allocation of personnel

          computer use

          reliability

          effects of programming methods

Quantitative measurement of programs, where the measurements can be related to intrinsic properties, has appeal from an engineering standpoint. Other engineering disciplines have constraints on design that can often be expressed numerically. The designer of circuit chips, for example, deals with technology limits

such as the number of access pins, the number of circuits that can be housed in a chip, and so forth. These limits are in turn derived from other limits, such as, heat dissipation, voltage limits, etc., that can also be dealt with quantitatively.

The measurement of programs is still a fairly subjective process. The easiest way the size of a program can be measured is based on the lines of code or number of statements, but acceptance of these measures is not universal. Another measure is measurement of program complexity, which some feel is related to the number of decision nodes in a program [McC76]. The problem is that both size and complexity are measured after the fact. That is, measurement is not possible until the code has been written. Elements of measurements can be considered if logic is outlined before the code has been written. Even then, measurements tend to be defense mechanisms against problems identified by other means, such as late schedules and high defect levels.

The theory of software science was developed by the late M.H. Halstead during the early 1970's. Halstead's development effort was mainly empirical. He measured a

number of characteristics and a number of properties. In his approach of measurement of software complexity, code is broken down into atomic particles of operators and operands. The basic metrics are:

n1 = number of unique operators

n2 = number of unique operands

N1 = total occurences of operators

N2 = total occurences of operands

f1,j = number of occurences of the jth most frequently occuring operator, where j = 1,2,...,n

f2,j = number of occurences of the jth most frequently occuring operand, where j = 1,2,...,n

Generally any symbol or keyword in a program that specifies an algorithmic action is considered as an operator, while a symbol used to represent data is considered as an operand. Punctuation marks are considered to be operators.

From the above basic metrics, the size of the vocabulary of a program is defined as

n = n1 + n2

The actual length, N, of a given program is defined as the sum of the total occurences of the operators and total occurences of the operands . This actual length is closely related to the traditional "Lines of Code"

measure of program length and is given by

$$N = N1 + N2$$

The unit for N is the number of tokens instead of number of lines.

The predicted length or the estimated length $\hat{N}$ of the computer program is given by

$$\hat{N} = n1 \log_2(n1) + n2 \log_2(n2)$$

Additional metrics are defined by Halstead using the basic terms n1, n2, N1, and N2. The volume V of a program is measured in bits. This is given by

$$V = N \log_2 n$$

The minimum possible volume for a given program is called the potential volume, V*. This is given by

$$V^* = (2 + n2^*) \log_2(2 + n2^*)$$

where n2* is the observed input/output operands required by the program.

Program level L is a measure of the succinctness of an implementation of an algorithm. It is defined as

$$L = V^*/V$$

where V* is the potential volume of the program. A program can be implemented by many different but equivalent programs, and that program that implements an algorithm in its most succicnt form has the largest implementation level.

Finally, Halstead derived relationships for measuring the effort and time required to generate a given program. Those expressions are

$$E = V/L$$

and

$$T = E/S$$

where E is the number of elementary mental discriminations required to generate a given program and S is an estimate of the number of such discriminations in unit time.

AIM OF STUDY

The model discussed here is a tool for validating Halstead's measures. The model is designed to evaluate the estimated length formula developed by Halstead. The model will be used to investigate the relationships among Halstead's parameters and with other metrics. However, The first step in experimentation is to calibrate the model. The aim of this study is to implement the model, study its basic behaviour, and study how the model relates to Halstead's length equation.

REVIEW OF RELATED LITERATURE

A large amount of work has been done in the last
few years in the field of software science. Although
the concept of software engineering has existed only for
two decades and software science for even less time,
much material has been written on software science
topics. While not exhaustive, the following
bibliography includes significant references:

[Aar85]	[Alb83]
[Bak79]	[Bak80]
[Bas83a]	[Bas83b]
[Beh83]	[Cur79a]
[Cur79b]	[Els76a]
[Feu79]	[Fit80]
[Fitz78]	[Hal77]
[Hal80]	[Han78]
[Las79]	[Old77]
[Old79]	[Ott76]
[She80]	[Zwe79]

LENGTH EQUATION

The definitive work on the origins of software
science appeared in Halstead's 1977 monograph "Elements
of Software Science" [Hal77]. He presented a number of
equations using counts of operators and operands to
predict a wide range of criteria. Halstead proposed
equations to calculate the actual length and the
estimated length of programs. The following are a list

8

of references that make use of Halstead's length eqautions:

    [Shen79]   [Smi80]
    [Aar85]   [Alb83]
    [Els76]   [Feu79]
    [Fitz78]   [Las79]
    [Zwe79]

The length equations are dependent on the number of unique operators and on the number of unique operands. One difficulty in using the length equation is in how to classify tokens into operators and operands. In the work done by Halstead [Hal77] most of the supporting data was drawn from algorithms written in Algol and Fortran. For these two languages, it did not seem very difficult to classify tokens into operators and operands. Variable declaration sections and other non-executable statements were excluded from the counts in computer programs. However, in other languages, it is sometimes impossible to determine whether a token is to be interpreted as an operator or operand, for example, (setq x 'sqrt) and (setq x (funcall x 16)) is a case of where 'x' can be treated as either an operator or an operand [Las81]. Since the variable declaration section in some languages (eg., data division in Cobol) represents a significant portion of the programming effort, it does not seem reasonable to ignore it

9

[Shen79], [Fit79], [Els78].

Another objection raised by Lassez was the ambiguity involved in the counting of the GO TO's and the IF statements in Fortran. Halstead suggested that each 'GO TO Label' be counted as a unique operator for each unique label. On the other hand, n IF statements are considered to be n occurences of one unique IF operator.

Moreover, work done by Shen [Shen83] and Smith [Smi80] showed that Halstead's estimated length equation did not hold for programs of all lengths. From their work, it was seen that Halstead's length equation over-predicted for small programs and under-predicted for large programs. However, the equation worked well for programs in the range of 2000 <= N <= 4000.

These ambiguities, like the counting of the GO TO's, the counting of the nested IF statements, and the classification of operators and operands depending on the language used, are some of the difficulties encountered in using Halstead's length equations.

EMPIRICAL WORK

Experiments have been conducted by Halstead and others to validate these software measures. Tests have

been conducted on PASCAL programs, PL/1 programs, software subsystems, FORTRAN programs, etc. Elshoff [Els76], Feuer [Feu79], Fitzsimmons [Fitz78], and Lassez [Las79] observed excellent correlation between predicted and observed program lengths. However, these works have been criticized on the following grounds [Shen83]:

1) Sample sizes were too small

2) Program sizes were too small

3) Many of the experiments, especially those concerning programming time, involved only single subjects

4) The subjects were generally college students

5) Halstead in his derivation of length equations gives no theoretical backing to some of his assumptions

THEORETICAL JUSTIFICATION

The software science theories originally proposed by Halstead have prompted extensive research by others. Woodfield [Woo79] and Baker and Zweben [Bak79] used software science measures in more extensive experiments to investigate problem and program complexity. Gordon [Gor79] studied program clarity through software science relationships. Woodfield and

11

Gordon each made significant use of Halstead's estimate of programming effort E. Curtis et al. [Cur79a] also investigated aspects of software complexity experimentally by contrasting Halstead's E measure, McCabe's complexity measure, and the length of the pertinent programs as measured by the number of statements. Pursuing research in somewhat a different direction Comer [Com79] argued that software science parameters are appropriate metrics in the study of the top-down design of programming projects. The measurements of the design process were done on a purely experimental basis using controlled conditions, i.e., the program was well defined, unambiguous, and independent of human talent. Ottenstein [Ott79] employed software measures to aid in predicting the number of bugs in a system at the beginning of the testing and integration phases of development. Most of these researchers have concentrated on experimentally testing those measures. They have, for most part, not addressed the theory behind those measures. Hence, the goal should be a set of measures that can be justified theoretically, that can be supported empirically, and that can be used with confidence by programmers and project managers.

SHOOMAN'S WORK

Shooman's work [Sho83] focussed on the basic probabilistic and information theoretic models. He related software science to basic probabilistic models by applying Zipf's law to computer programs. Using Zipf's law, he derived an equation that, given the number of types of words used in a computer program, could estimate the length of the program, ie.,

Length = $n = t(0.5772 + \ln(t))$

where 't' is the number of word types used.

Shooman views the program as a string of tokens. The token string which represents the program is generated by choosing an operator token at random from the set of operators, then choosing an operand token at random from the set of operands, and continuing this alteration process. The program generation stops when the last unused operator or operand token is chosen for the first time. Based on this and basic statistical theory, he derives an expression for the sequence length, which is given by:

$$E(SL_n) = n \sum_{k=1}^{n} 1/(n-k+1) = n \sum_{i=1}^{n} 1/i \qquad (1)$$

By making an asssumption $i = 2^j$ he derives the equation

13

$$E(SL_n) = n \sum_{j=0}^{\log_2 n} 1/(2^j) \qquad (2)$$

and by assuming that $1/(2^j) \leq 1$ he says

$$E(SL_n) \leq n\log_2 n$$

The work done by Shooman is questionable. Equations (1) and (2) are not equal since the expansion of those equations do not yield equal results. Moreover, Shooman's work has been criticized extensively by Moranda [Mor85], on grounds of meaningless substitutions, equating different proportionality constants, alteration of source data set, and violation of Zipf's law.

14

THE MODEL

The basic model of the program is as shown in fig.1. The model is a bipartite digraph [Joe83]. Each node in the model is either
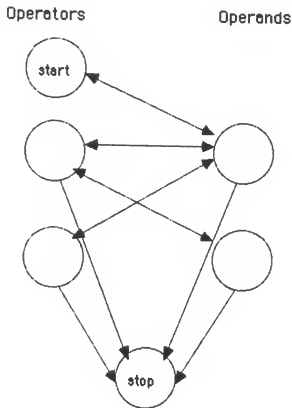
Fig. 1. Bipartite Digraph

15

an operator or an operand. The bipartite nature of the model depicts the assumption of Halstead that operators and operands alternate in a program. The digraph is complete, in that, from a node in one section of the graph there is an arc to every other node in the other section. This completes the idea that an operator comes after an operand and vice-versa.

One of the nodes is identified as the start node and another node is identified as the terminal or stop node. Both these nodes, i.e., the start node and the terminal node, are also considered to be operators. There are no arcs out of the terminal node, and hence, the terminal node acts as a sink.

Transition probabilities are assigned to each arc in the graph.

$Pt(k,j)$: the probability of transition from node 'k' to node 'j'.

If the kth node represented the operator '+' and the jth node represented the operand 'x' then $Pt(k,j)$ represents the probability of 'x' coming right after '+'.

$Pt(k,k) = 0$ for all k : this means that no operator or operand can follow itself.

$P(j,i)$: the probability of visiting a node 'j' on the 'i'th iteration.

16

In terms of program terminology this means that if the
jth node represents an operator '+', then P(j,i)
represents the probability of being in '+' after 'i'
iterations.

Pv(j,i) : the probability of having visited a node
          'j' during 'i' iterations.
If the jth node represents an operator '+', then Pv(j,i)
represents the probability of having visited '+' before
or during the ith iteration.

The probability of being in a node 'k'after 'i'
iterations is the sum for all nodes 'j' of the
probability of being in a node 'j' after (i-1)
iterations times the probability of transitioning from
node 'j' to node 'k'. This is given as

$$P(k,i) = \sum_j (P(j,i-1) * Pt(j,k))$$

The probability of having visited a node 'k' during 'i'
iterations is the probability of having visited that
node during (i-1) iterations plus the probability of
not having visited that node during (i-1) iterations
times the probability of being in that node at the end

of the 'ith' iteration. This is represented as

$$Pv(k,i) = Pv(k,i-1) + (1-Pv(k,i-1)) * Pt(k,i) \quad !$$

The expected length is denoted by E(length) or El and expected nodes is represented by E(nodes) or En.

Since the model is not closed, i.e., the terminal node acts as a sink the probabilities decrease as the number of iterations increase. The expected value of the length is the sum for each iteration of that iteration times the probability of reaching the stop node 'z' for that iteration. This is represented as

$$El = \sum i * P(z,i)$$

TESTS AND RESULTS

The above model was implemented using the language 'C'. Appendix A shows the module hierarchy of the implemented model and the listing of the implemented model. The OS used was UNIX and the hardware used was VAX-11/780.

! Recent discussions indicate this may be slightly too large due to lack of independence between events.

The study of the model can be best described in terms of the history of the work done. Opd is the number of operand nodes in the model. Opr is the number of operator nodes. Opr includes the start and the stop node. Initially, the model was run with

```
Pt(operators to terminal node)  = 0
Pt(operands to terminal node)   = 1/Opd
Pt(operators) = 1/(Opr  - 1)
Pt(operands)  =  (1 - 1/Opd)/Opd
```

The model was tested for runs in which (operators) Opr < (operands) Opd, Opr > Opd, and Opr = Opd. Table 1 shows the results of these runs. Graph 1 was plotted for the sum of operators and operands for the condition Opr = Opd versus the estimated length from the model. The key point of interest was the fact that for high values of Opr and Opd the estimated length seemed to follow a curve instead of a straight line.

Initially, this was attributed to the fact that Pt(operators to terminal node) = 0.0. So the model was modified to accomodate the conditions

```
Pt(terminal  node) <> 0.0.
Pt(Opr)  =  (1 - Pt(terminal node))/(Opr - 1)
```

```
        [Opr,Opd]          Est. Length
                            from Model

--------------------------------------

                Opr > Opd

     5,3                      9.73
    20,8                     14.31
    10,8                     14.58
    15,5                     15.11
    25,5                     13.21
    30,10                    12.13

                Opr < Opd

     3,5                      6.00
     8,20                    13.40
     8,10                    13.40
     5,15                     9.73
     5,25                     9.73
    10,30                    14.58

                Opr = Opd

     3,3                      6.00
     5,5                      9.73
    10,10                    14.58
    15,15                    15.11
    20,20                    14.31
    25,25                    13.21
    30,30                    12.13
    40,40                    10.29
    50,50                     8.86
```

Table 1. Estimated length from the model for
        Pt(Operators to halt) = 0.0
        Pt(Operands to halt) = 1/Opd
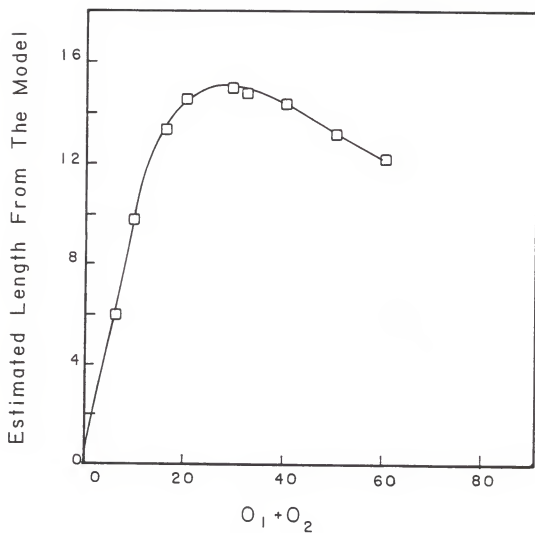
Graph 1. Estimated length from the model
versus sum of Opr and Opd

Pt(Opd) = (1 - Pt(terminal node))/Opd

The model was rerun again with the same values as shown in Table 1. The results of this test are shown in Table 2. Once again it was seen that as the sum of the operators and operands increased the estimated length calculated from the model seemed to follow a smooth curve. It was now that we realized that the model was not being run long enough. In fact, the model was executing for 50 iterations. This caused still significant probabilities of not having terminated. Hence from here on the model was run to the limit of

# of iterations * Pt(stop node) <= (2 * 10**-5)

Once the limit to which the model was to be run was established, the next step was to study how the model behaved in prediction of length when subjected to changes in operators and operands. In order to study this the model was run for [operators,operands] being [x,5], [x,15], [10,x], and [20,x] where 'x' is an integer value being either an operator or an operand depending on the nature of the run. Table 3 shows the results of these runs. The model conditions for these runs were

```
        [Opr,Opd]              Est. Length
                                from Model

    -----------------------------------

                Opr > Opd

      5,3                          7.92
     20,8                         15.04
     10,8                         13.93
     15,5                         14.41
     25,5                         15.00
     30,10                        14.76

                Opr < Opd

      3,5                          7.92
      8,20                        15.04
      8,10                        13.93
      5,15                        14.41
      5,25                        15.00
     10,30                        14.26

                Opr = Opd

      5,5                          9.66
      8,8                         13.25
     10,10                        14.41
     15,15                        15.00
     16,16                        14.91
     20,20                        14.26
```

Table 2. Estimated length from the model for
         Pt(to halt) <> 0.0

[Opr,Opd]	Est. Length from Model	[Opr,Opd]	Est. Length From Model
3,5	7.92	10,5	14.83
10,5	14.83	10,8	17.79
15,5	19.77	10,10	19.77
20,5	24.72	10,15	24.72
25,5	29.66	10,20	29.66
30,5	34.60	10,25	34.60
35,5	39.55	10,30	39.55
40,5	44.49	10,35	44.49
3,15	17.79	20,3	22.74
5,15	19.77	20,5	24.72
10,15	24.72	20,8	27.68
15,15	29.66	20,10	29.66
20,15	34.60	20,15	34.60
25,15	39.55	20,20	39.55
30,15	44.49	20,25	44.49
35,15	49.42	20,30	49.42
40,15	54.37	20,35	54.37

Table 3. Estimated length from the model for
          Pt(halt node) = 1/(Opr + Opd)

```
Pt(terminal node) = 1/(Opr + Opd)
Pt(Opr)  = (1 - Pt(terminal node))/(Opr - 1)
Pt(Opd)  = (1 - Pt(terminal node))/(Opd)
```

From these runs it was seen that the estimated length
from the model was almost equal to Opr + Opd. It was
also seen that irrespective of the values of operator or
operand as long as the sums of the two remained the same
the estimated length from the model remained the same.
This meant that the length estimated from the model was
either dependent on the sum of operators and operands,
or, it could be dependent on the probability of
transition to the stop node. To check if either of the
above was true or not the model was modified for the
following conditions:

```
Pt(terminal node ) = 1/K.(Opr + Opd) where K = 1...n
Pt(Opr) = (1 - Pt(terminal node))/(Opr -1)
Pt(Opd) = (1 - Pt(terminal node))/Opd
```

Under these conditions the estimated length from the
model did not match the length established in Table 3.
This meant that the model was dependent on the
probability of transition to the stop node.

    The next step in the study of the model was to see
if the model was dependent on the positional occurence

of the programming statements. For this we took as an example, the Pascal language and counted the number of operators in the Pascal language. This amounted to 40. The assumed number of operands for this study was taken to be 80. The model was run under the following conditions:

Pt(terminal node) = 1/(Opr + Opd)
Pt(changed node in operands) = x where x is a value < 1
Pt(remaning operands) = (1-Pt(terminal node)- x)/(Opd-1)
Pt(changed node in operators) = x
Pt(remaining operators) = (1-Pt(terminal node)-x)/(Opr-2)

From this study it was seen that the model came up with the same estimated lengths for every run, which means that the model is not dependent on the likelihood of different types, e.g., if 'WHILE' operators are twice as common as 'IF'. Also it was studied that the model was independent of individual transitions to the stop node and was dependent on the average probability of transition to halt.

The last phase of our study focussed on whether the estimated length from the model matched Halstead's length equation, and if so derive an equation to produce any length for a given Opr and Opd by changing the

26

transition to the halt node.   The following conditions
held for the model

```
 Pt(terminal nodes) <> 0.0
 Pt(operators) = (1 - Pt(terminal node))/(Opr - 1)
 Pt(operands)  = (1 - Pt(terminal node))/Opd
```

The model was run for Opr,Opd being [40,80], [30,30],
[45,15], and [5,10].   The results are shown in Table 4
and the plot of these results in Graph 2.   From this
study it was seen that the model matched Halstead's
length equation at one point, the point being the one
where the system was just about to be overloaded.
Overloading of the system occured when the difference
between Opr and estimated operators, and Opd and
estimated operands were both less than one.   The
estimated length from the model was a straight line.
The model was analyzed by Drs. Mark and Sally McNulty
from the Statistical Department.   They derived an
equation for the expected length given the transition to
the stop node.   The derivation of the equation is shown
in Appendix B. The derived equation for expected length
is given as

$$E(1) = \frac{2P2(1 - P1) + P1(1 + (1 - P1)(1 \_ P2))}{[1 - (1 - P1)(1 - P2)]^{**2}}$$

Pt(Opr to halt) + Pt(Opnd to halt)	Est. Length from Model	Halstead's Length for (40,80)	Halstead's Length for (80,40)	Halstead's Length for (30,30)	Halstead's Length for (45,15)	Halstead's length for (5,10)
6E-4	3332.61	716.24	715.77	292.08	303.19	43.46
8E-4	2500.00	716.14	715.77	292.08	303.19	43.46
9E-4	2212.00	716.13		292.08	303.19	
1E-3	2000.00	716.13	715.76	292.08	303.19	43.46
1.2E-3	1661.00	716.13		292.08	303.19	
1.6E-3	1250.00	716.12		292.08	303.19	
2.0E-3	1000.00	714.96	714.70	292.08	303.18	43.46
2.7E-3	740.37	710.11	710.12	292.08	303.12	43.46
6.0E-3	332.76	636.02	638.78	292.08	303.12	43.46
7.0E-3	285.63	609.57	610.30	290.81	296.48	43.46
8.0E-3	250.00	580.88	581.72			
1.25E-2	159.51	464.39	466.60			43.33
1.75E-2	113.97	370.77	373.30	240.02	221.92	42.96
2.25E-2	88.72	303.75	306.03	212.44	194.22	42.30
2.75E-2	72.50	254.88	256.43	187.96	171.37	41.37
3.25E-2	61.39					39.03
4.25E-2	46.77	165.98	166.13			37.88
4.75E-2	41.98	147.17	148.30	122.01	112.51	36.40
5.25E-2	37.70	131.06	131.53			
5.75E-2	34.33					
6.25E-2	31.34	106.25	107.34	92.16	86.31	33.76

Table 4. Comparison of Estimated length from the Model with Halstead's length for different Opr and Opnd
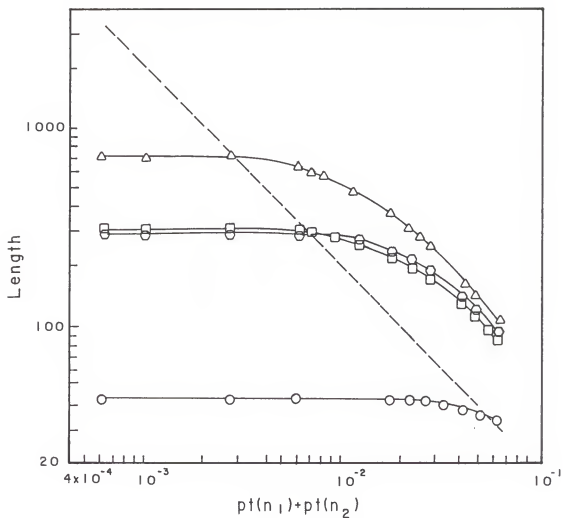
Graph 2. Estimated length from the model versus
Pt(Opr) + Pt(Opd)
△ Halstead's length for (40,80)
▢ Halstead's length for (45,15)
⊖ Halstead's length for (30,30)
○ Halstead's length for (5,10)

where

    P1 is the average probability of transition from the operators to halt

    P2 is the average probability of transition from the operands to halt.

This formula and derivation confirms our experimental result that the expected length depends only on the probability of transition to halt. In programming terms, this means that the length of a program does not depend on the number of operators or operands used but does depend on the probability of halting. Unfortunately, we have not yet found a way to estimate these probabilities from program characteristics.

The values for $Pv(j,-)$ which represent the number of unique operators or operands do depend on all the values in the transition matrix and not just the probability of transition to the terminal state. Thus, the number of unique operands and operators does not appear to be sufficient for predicting the length of the program.

The model that we have studied about is still in its infant stage of development. Since very little was known about the model most of the model studies had to

be conducted on a trial and error basis. Modifications to the model were relatively easy, but, the times taken for individual runs were exceptionally large, sometimes reaching 48 hours. From our studies, the model seems to have the potentiality in settling debates over the counting rule used, in developing insight into the effects of the language on the size and complexity of a program, relationships between the module properties and programmer style, etc.

CONCLUSIONS AND FUTURE WORK

CONCLUSIONS

-- The first conclusion established from the study is that the expected length is not dependent on the likelihood of different types.

-- The second conclusion established was that the model was independent of individual transitions to the stop node and was dependent on the average probability of transition to halt.

-- The model matched Halstead's length equation at one point, the point being the one where the system was just about to be overloaded. This means that Halstead's length equation is a special case of the model.

-- A mathematical derivation was established to calculate the expected length from the model based on the transition to halt.

FUTURE WORK

The bipartite digraph model described in this paper is a first step in establishing a theoretical foundation for Halstead's metrics. Future work can be directed in

-- Studying the models behaviour in predicting

estimated nodes visited and trying to establish a mathematical equation in effect of predicting estimated nodes visited depending on the transition to the halt.

-- Study the model and its relationships to Halstead's metrics.

-- Study how a programmers style affects the length predictions.

-- Study how the syntax of a particular language affects the model. This can be done by relaxing the bipartite nature of the model.

REFERENCES

[Aar85]   Aaron H. Kouston and Donald E. Wood,
             "Software Science Applied to APL," IEEE
             Trans. Software Eng., vol. SE-11, No. 10,
             Oct. 1985, pp. 994-1000.

[Alb83]   A. J. Albrecht and J. E. Gaffney, Jr.,
             "Software Function, Source Lines of Code,
             and Development Effort Prediction: A
             Software Science Validation," IEEE Trans.
             Software Eng., vol. SE-9, No. 6, March
             1983, pp. 639-647.

[Bak79]   Albert L. Baker and Stuart H. Zweben, "The
             Use of Software Science in Evaluating
             Modularity Concepts," IEEE Trans. Software
             Eng., vol. Se-5, no. 2, March 1979, pp.
             110-120.

[Bak80]   Albert L. Baker and Stuart H. Zweben, "A
             Comparison of Measures of Control Flow
             Complexity," IEEE Trans. Software Eng.,
             vol. SE-6, no. 6, November 1980, pp. 506-
             512.

[Bas83a]   V. R. Basili, R. W. Selby, and T. Y.
             Phillips, "Metric Analysis and Data
             Validation," IEEE Trans. Software Eng.,
             vol. SE-9, no. 6, March 1983, pp. 652-663.

[Bas83b]   V. R. Basili and D. H. Hutchens, "An
             Empirical Study of a Systematic Complexity
             Family," IEEE Trans. Software Eng., vol.
             Se-9, no. 6, March 1983, pp. 664-672.

[Bau72]   F. L. Bauer, "Software Engineering,"
             Information Processing 71 (Amsterdam:
             North Holland Publishing Co., 1972), pp.
             530.

[Beh83]   C. A. Behrerns, "Measuring the
             Productivity of Computer Systems," IEEE
             Trans. Software Eng., vol. SE-9, no. 6,
             March 1983, pp. 649-651.

[Boe76]   Boehm B.W. "Software Engineering". IEEE

<u>Transaction on Computers</u>, vol. C-25, no. 12, Dec. 1976, 1226-1241.

[Boe78]   Barry W. Boehm, John R. Brown, Hans Kaspor, Myron Lipow, Gordon J. MacLeod, and Michael J. Merritt, "Characteristics of Software Quality", North Holland, 1978.

[Chr81]   Christensen K., G.P. Fitsos, and C.P. Smith. "A Perspective on Software Science". <u>IBM Journal</u> vol. 20, no. 4 1981, 372-381.

[Com79]   D. Comer and M. H. Halstead, "A Simple Experiment in Top-Down Design," <u>IEEE Trans. Software Eng.</u>, vol. SE-5, March 1979, pp. 105-109.

[Cou83]   N. S. Coulter, "Software Science and Cognitive Psychology," <u>IEEE Trans. Software Eng.</u>, vol. SE-9, no. 6, March 1983, pp. 166-171.

[Cur79a] Bill Curtis, Sylvia B. Sheppard, Phil Milliman, M. A. Borst, and Tom Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with Halstead and McCabe Metrics," <u>IEEE Trans. Software Eng.</u>, vol. SE-5, no. 2, March 1979, pp. 96-104.

[Cur79b] Bill Curtis, Sylvia B. Sheppard, and Phil Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," <u>4th International Conference on Software Engineering</u>, September 1979, pp. 356-360.

[Els76]   James L. Elshoff, "Measuring Commercial PL/1 Programs Using Halstead's Criteria," <u>SIGPLAN Notices</u>, vol. 11, no. 5, May 1976, pp.38-46.

[Els78]   J. L. Elshoff, "An Investigation Into the Effect of Counting Method Used on Software Science Measurements," <u>ACM SIGPLAN Notices</u>, vol. 13, Feb. 1978, pp. 30-45.

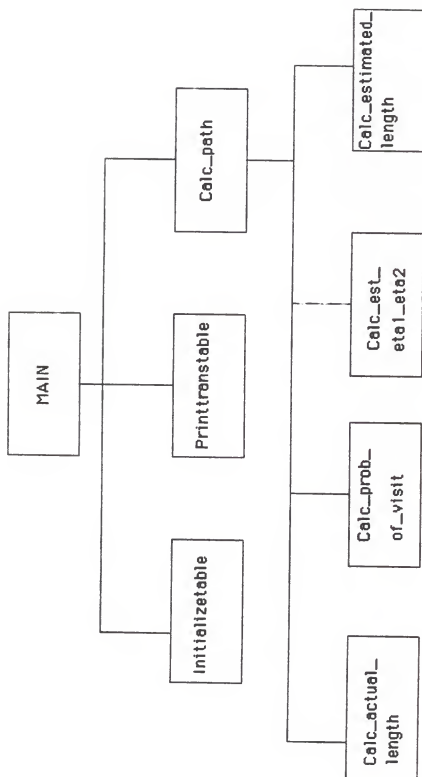[Feu79]   Alan R. Feuer and Edward B. Fowlkes, "Some

Results From an Empirical Study of Computer Software," <u>4th International Conference on Software Engineering</u>, September 1979, pp. 351-355.

[Fit79]   G. P. Fitsos, "Software Science Counting Rules and Tuning Methodology," <u>IBM Santa Teresa Lab.</u>, Tech. Rep. 03.075, Sept. 1979.

[Fit80]   George P. Fitsos, "Vocabulary Effects in Software Science," <u>COMPSAC 80</u>, Chicago October, 1980, pp. 751-756.

[Fitz78]   Ann Bowman Fitzsimmons and Tom Love, "A Review and Evaluation of Software Science," <u>Computing Surveys</u>, vol. 10, no. 1, March 1978, pp. 3-18.

[Gor79]   R. D. Gordon, "Measuring Improvement in Program Clarity," <u>IEEE Trans. Software Eng.</u>, vol. SE-5, no. 5, March 1979, pp. 24-25.

[Hal77]   M. H. Halstead, "Elements of Software Science," New York: Elsevier, North Holland, 1977.

[Hal80]   M. H. Halstead and Victor Schneider, "Self-Assesment Procedure VII," <u>Communications of the ACM</u>, vol. 23, no.8 August 1980, pp. 475-480

[Han78]   Wilfred J. Hansen, "Measurement of Program Complexity by the Pair (Cyclomatic Number Operator Count)," <u>SIGPLAN Notices</u>, vol. 13, no. 3, March 1978, pp. 29-33.

[Joe83]   Joe L. Mutt, Abraham Kandel, and Theodore P. Baker, "Discrete Mathematics for Computer Scientists," Reston Publishing Co., Inc., Reston, Virginia, 1983, pp. 400.

[Las79]   J. L. Lassez and D. van der Knijff, "Evaluation of Length and Level of Simple Program Schemes," <u>Compsac 79</u>, November 1979, pp. 688-694.

[Las81]   J. Lassez, D. Van Der Knijff, and J.
            Shepherd, "A Critical Examination of
            Software Science," J. Syst. Software, Vol.
            2, Dec. 1981, pp. 105-112.

[McC76]   McCabe T.J. "A Complexity Measure". IEEE
            Transactions on SE, vol. SE-2, Dec. 1976,
            308-320.

[Mor85]   P. Moranda, "Software Engineering:
            Relaibility, Development, and Management,"
            Computing Reviews, vol. 26, no. 2,
            February 1985, pp. 92-93.

[Old77]   R. R. Oldhoft, "A Contrast Between
            Language Level Measures," IEEE Trans.
            Software Eng., vol. SE-3, no. 6, November
            1977, pp. 476-478.

[Ott76]   J. Ottenstein, "An Algorithmic Approach to
            the Detection and Prevention of
            Plagarism," SIGCSE Bulletin, vol. 8, no.
            4, December 1976, pp. 30-41.

[Ott79]   L. M. Ottenstein, "Quantitaive Estimates
            of Debugging Requirements," IEEE Trans.
            Software Eng., vol. SE-5, no. 5, September
            1979, pp. 505-514.

[She80]   Sylvia B. Sheppard, Phil Milliman, and
            Bill Curtis, "Experimental Evaluation of
            On-Line Program Construction," Compsac 80,
            Chicago, October 1980, pp. 505-514.

[Shen79]   V. Y. Shen and H. E. Dunsmore, "Analyzing
            Cobol Programs via Software Science," IEEE
            Trans. Software Eng., vol. SE-5, March
            1979, pp. 79-90.

[Shen83]   Vincent Y. Shen, Samuel D. Conte, and H.
            E. Dunsmore, "Software Science Revisited:
            A Critical Analysis of the Theory and Its
            Empirical Support," IEEE Trans. Software
            Eng., vol. SE-9, March 1983, pp. 155-165.

[Sho83]   Martin L. Shooman, "Software Engineering:
            Reliability, Development, and Management,"
            McGraw-Hill, Inc., New York 1983.

[Smi80]   C. P. Smith, "A Software Science Analysis
            of Programming Size," <u>Proc. ACM Nat.</u>
            <u>Comput. Conf</u>., Oct. 1980, pp. 179-185

[Woo79]   S. N. Woodfield, "An Experiment in Unit
            Increase in Problem Complexity," <u>IEEE</u>
            <u>Trans. Software Eng</u>., vol. SE-5, March
            1979, pp. 76-84.

[Zwe79]   Stuart H. Zweben and Kin Chee Fung,
            "Exploring Software Science Relations in
            COBOL and APL," <u>Compsac 79</u>, Chicago,
            November 1979, pp. 702-707.

Appendix A

Module Hierarchy for Implemented Model

Boxes shown in the hierarchy:

- MAIN
  - Initializetable
  - Printtranstable
  - Calc_path
    - Calc_actual_length
    - Calc_prob_of_visit
    - Calc_est_eta1_eta2
    - Calc_estimated_length

```c
#include <stdio.h>
#include <math.h>
#define ETA 300
#define steps 50


/****************************************/
/* The main calls the procedures        */
/* "intialize", "printtranstable", and  */
/* "print_calc_path". However, before    */
/* it could execute these procedures     */
/* it prompts the user to enter the      */
/* of OPERATORS, OPERANDS, the Pt(halt)  */
/* from operators and Pt(halt) from      */
/* operands.                             */
/****************************************/


main ()

{
float trans[ETA][ETA];
int eta1, eta2, i, j, temp;
float trans_1, trans_2;
printf("\n Enter the number of OPERATORS inclusive of start
       and stop\n");
scanf("%d", &eta1);
printf("\n Enter the number of OPERANDS\n");
scanf("%d", &eta2);
printf("\n Enter the prob. of transition to operator stop
       node\n");
scanf("%f", &trans_1);
printf("\n Enter the prob. of transition to operand  stop
       node\n");
scanf("%f", &trans_2);
initialize(trans, eta1, eta2, trans_1, trans_2);
print_calc_path(trans,eta1,eta2,trans_1,trans_2);
}

/****************************************/
/* This proceedure "initialize" initia- */
/* lizes the matrix trans depending on  */
/* the number of OPERATORS(eta1) and    */
/* the number of OPERANDS (eta2).       */
/* NB: There should be a minimum number */
/* of 3 OPERATORS and at least 1 OPERA- */
/* ND. OPERATORS include terminals      */
```

```
/* "start" and "stop".                  */

initialize(a,n1,n2,t1,t2)

float a[ETA][ETA];
int n1, n2;
float t1,t2;
{
int l, j;
for (l=0; l<n1-1; l++)
      {
      for (j=0; j<n1-1; j++)
                  {
                  a[l][j] = 0;
                  }
      }

for (l=0; l<n1-1;l++)
      {
      j=n1-1;
      a[l][j] = t1;
      }

for (l=0; l<n1-1; l++)
      {
      for (j=n1; j<n1+n2; j++)
                  {
                  a[l][j] = (1.0 - t1)/n2;
                  }
      }

l=n1-1;
for (j=0; j<n1+n2; j++)
      {
      a[l][j] = 0;
      }

for (l=n1; l<n1+n2; l++)
      {
      for (j=0; j<n1-1; j++)
                  {
                  a[l][j] = (1.0 - t2)/(n1-1);
                  }
      }

for (l=n1; l<n1+n2; l++)
      {
      j=n1-1;
```

```
        a[l][j] = t2;
        }

for (l=n1; l<n1+n2; l++)
      {
      for (j=n1; j<n1+n2; j++)
                  {
                  a[l][j] = 0;
                  }
      }
}
/*****************************************/
/* This proceedure "printtranstable"     */
/* prints out the matrix "trans" which   */
/* was initialized by the proceedure     */
/* "initialize".                         */
/*****************************************/

printtranstable(a,n1,n2)

int n1, n2;
float a[ETA][ETA];

{
int x, y;
int temp;

for (x=0; x<n1+n2; x++)
     {
     printf("\n Row = %d\n",x);
     temp = 0;
     for (y=0; y<n1+n2; y++)
               {
               if (temp>7)
                  {
                  temp = 0;
                  printf("\n");
                  }
               printf(" %f ",a[x][y]);
               temp++;
               }
     }
}


/*****************************************/
/* This proceedure "print_calc_path" does*/
```

```c
/* two primary functions. It calculates  */
/* the paths and then prints it out.     */
/* The paths are calculated by taking the*/
/* the sigma of the product of each      */
/* element of array "b[]" with each row   */
/* element of the matrix "a[][]".        */
/* The sigma of each                      */
/* row becomes the individual element of  */
/* the new b[]. The calculation is thus   */
/* repeated fifty times.                  */
/*****************************************/


print_calc_path(a,n1,n2,t1,t2)

int n1,n2;
float t1, t2;
float a[ETA][ETA];

{
float b[ETA];
float sum;
float est_num_nodes;
float c[ETA];
float visit[ETA]; /* is used for calculating nodes visited*/
int x, y, z, count;
float actual; /* This is used for calculating the actual */
        /* length.  It is passed to        */
        /* "calc_actual_length".           */

double est_n1;/* Is used to calculate estimated operators.*/
        /* It is modified at "calc_est_eta1_eta2" */
        /* and the modified value is then passed to */
        /* "calc_estimated_length" where it is used */
        /* in the formula : est_length = n1log(n1) */
        /* + n2log(n2).  The log is to the base 2. */
        /* log to the base 2 is same as log(n1) to */
        /* the base 10 divided by log(2) to        */
        /* the base 10. This is the calculation    */
        /* which is used in "calc_estimated_length". */

double est_n2;/* The same as above but applied to est_n2 */
double est_length;

int temp;

printf("\n\n\n\n\n");
printf("STATISTICS OF THE PATHS AND VISITS\n");
printf("--------------------------------\n");
```

```
printf("\n\n");

actual = 0;
est_n1 = 0;
est_n2 = 0;
count = 0;
b[0] = 1;
visit[0] = 0;
for (x=1; x<ETA; x++)
     {
     b[x] = 0;
     visit[x] = 0;
     }

temp = 0;
for (z=0; z<n1+n2; z++)
     {
     if (temp>7)
        {
        temp = 0;
        }
        temp++;
     }

calc_actual_length(b,&actual, count,n1);
est_num_nodes = 0.0;
print_calc_prob_of_visit(visit,b,n1,n2,count,
                    &est_num_nodes);
calc_est_eta1_eta2(visit,b,n1,n2, &est_n1, &est_n2);
est_length = 0.0;
calc_estimated_length(&est_n1,&est_n2,&est_length);


for (count=1;; count++)
 {
 for (x=0; x<n1+n2; x++)
         {
         sum = 0;
         for (y=0; y<n1+n2; y++)
                 {
                 sum = sum + (b[y] * a[y][x]);
                 }
         c[x] = sum;
         }
 for (z=0; z<n1+n2; z++)
         {
         b[z] = c[z];
         }
```

```
temp = 0;
for (z=0; z<n1+n2; z++)
        {
        if (temp>7)
            {
            temp = 0;
            }
            temp++;
        }

        calc_actual_length(b,&actual,count,n1);
        est_num_nodes = 0.0;
        print_calc_prob_of_visit(visit,b,n1,n2,
                            count,&est_num_nodes);
        calc_est_eta1_eta2(visit,b,n1,n2,
                        &est_n1, &est_n2);
        est_length = 0.0;
        calc_estimated_length(&est_n1,&est_n2,
                            &est_length);
        if ((count*b[n1-1]) <= 0.00002)
            {
        printf(" Product          = %f\n",
                                count*b[n1-1]);
        printf(" Operators        = %d\n", n1);
        printf(" Operands         = %d\n", n2);
        printf(" 1/operators      = %f\n",(t1));
        printf(" 1/operands       = %f\n",(t2));
        printf(" Est. Operators   = %f\n",est_n1);
        printf(" Est. Operands    = %f\n",est_n2);
        printf(" Actual length    = %f\n",actual);
        printf(" Estimated length = %f\n",est_length);
        printf(" Path length      = %d\n",count);
        printf(" Est_num_nodes    = %f\n",est_num_nodes);
        break;
            }
  }
}


/********************************************/
/* This procedure "print_calc_prob_of_visit*/
/* calculates the statistics for the prob- */
/* ability of visiting a certain node for a*/
/* certain path length. It then prints it  */
/* out and the calculates the number of    */
/* nodes it could visit for that path      */
/* length.                                 */
/********************************************/
```

```
print_calc_prob_of_visit(aa,bb,nn1,nn2,ccount,num)


float aa[ETA]; /* same as visit[ETA] from caller */
         /* print_cala_path */
float bb[ETA]; /* same as b[ETA] from caller */
         /* print_calc_path */
int nn1, nn2;  /* same as n1, n2 resp. from */
         /* print_calc_path */
int ccount;    /* same as count form caller */
         /* print_cala_path */
float *num;    /* same as est_num_nodes */

{
int  i;
float newvisit[ETA];
int temp;

for (i=0; i<nn1+nn2; i++)
    {
    newvisit[i] = aa[i] + (1 - aa[i]) * bb[i];
    aa[i] = newvisit[i];
    }
temp = 0;
for (i=0; i<nn1+nn2; i++)
    {
    if (temp>7)
       {
       temp = 0;
       }
       temp++;
    *num = *num + aa[i];
    }

}


/***********************************************/
/*   This proceedure "calc_actual_length"    */
/* calculates the actual length for Halsteads*/
/* formula. The formula used here is:        */
/* actual = actual + (count * P[z,i]).       */
/* actual and count are variables from the   */
/* procedure "print_calc_path".              */
/***********************************************/
```

```
calc_actual_length(bb,aactual,ccount,nn1)


float bb[ETA];
float *aactual;  /* same as actual from "print_calc_path" */
int ccount;      /* same as count  from ..............  */
int nn1;         /* same as n1     from ..............  */

{
float i;
i = *aactual;


i = i + (ccount * bb[nn1-1]);
*aactual = i;
}


/***********************************************/
/* This proceedure calculates the estimated   */
/* no of operators and operands. The formula  */
/* used for this is as follows:               */
/* est_nn1 = (sigma(sigma of visit for        */
/*            operators alone * probability    */
/*            of reaching the stop node))      */
/* est_nn2 = same as above but for operands.  */
/* These variables are then passed to         */
/* "calc_estimated_length".                   */
/***********************************************/


calc_est_eta1_eta2(vvisit,bb,nn1,nn2,est_nn1,est_nn2)


float vvisit[ETA]; /* same as visit[ETA]   */
float bb[ETA];     /* same as b[ETA]       */
int nn1, nn2;      /* same as n1 and n2    */
double *est_nn1;   /* same as est_n1       */
double *est_nn2;   /* same as est_n2       */

{
int i;
double sigma_visit_eta1;
double sigma_visit_eta2;

sigma_visit_eta1 = 0;
sigma_visit_eta2 = 0;
```

48

```
for (i=0; i<nn1; i++)
    {
    sigma_visit_eta1 = sigma_visit_eta1 + vvisit[i];
    }

*est_nn1 = sigma_visit_eta1;

for (i=nn1; i<nn1+nn2; i++)
    {
    sigma_visit_eta2 = sigma_visit_eta2 + vvisit[i];
    }

*est_nn2 = sigma_visit_eta2;
}


/**********************************************/
/* This proceedure calculates the estimated */
/* length using Halsteads equation. The     */
/* equation used is as follows:             */
/* N hat = n1 log(n1) + n2 log(n2). The log */
/* is to the base 2.                        */
/**********************************************/


calc_estimated_length(est_nn1,est_nn2,est_llength)

double *est_nn1;
double *est_nn2;
double *est_llength;

{
double const ;
double temp1;
double temp2;
double i, j;
double zero;

zero  = 0.0;
const = 2.0;
i = *est_nn1;
j = *est_nn2;

if ( i>zero && j>zero)
{
temp1 = ( i * (log10(i)/log10(const)));
temp2 = ( j * (log10(j)/log10(const)));
```
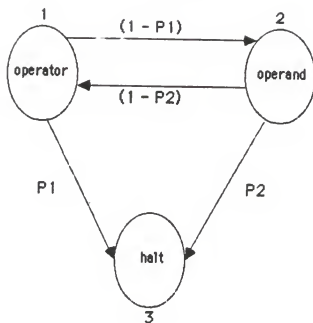
49

```
*est_llength = temp1 + temp2;
}
}
```

Appendix B

P1 is average probability of transition from the operator to halt
P2 is average probability of transition from the operand to halt

Expected length = E1 = Expected(length for number of iterations
being odd)

+

Expected(length for number of iterations
being even)

Expected length for number of iterations being odd

For number of iterations being odd

```
(1) -----> (3)               no. of iterations = 1
(1) --> (2) --> (1) --> (3)  no. of iterations = 3
```

Therefore probability that number of iterations is odd

i.e., $P(1_{odd}) = (1 - P1)^{(n-1)/2} (1 - P2)^{(n-1)/2} P1$

$$\text{where } n = 1, 3, 5, \ldots, n$$

Therefore $E(1 \text{ for } 1_{odd}) = \sum_{i=0}^{\infty} (2i + 1) \ [(1 - P1)(1 - P2)]^i \ P1$

Expanding the R.H.S., the R.H.S

$$E(1 \text{ for } 1_{odd}) = \sum_{i=0}^{\infty} 2P1i[(1 - P1)(1 - P2)]^i + \sum_{i=0}^{\infty} P1 \ [(1 - P1)(1 - P2)]^i$$

$$= \frac{(2P1)}{[1-(1-P1)(1-P2)]^2} \sum_{i=0}^{\infty} i \ [(1-P1)(1-P2)]^i \ [1-(1-P1)(1-P2)]^2$$

$$+$$

$$\frac{P1}{[1-(1-P1)(1-P2)]} \sum_{i=0}^{\infty} [(1-P1)(1-P2)]^i \ [1-(1-P1)(1-P2)] \quad \blacksquare$$

53

Consider the second summation to infinity term of equation ∎

$$= [1-(1-P1)(1-P2)] \sum_{i=0}^{\infty} [(1-P1)(1-P2)]^i \qquad ∎_2$$

Let $(1-P1)(1-P2) = R$

Therefore term $∎_2 = (1-R) \sum_{i=0}^{\infty} a R^i$

but $\sum_{i=0}^{\infty} R^i$ is of the general form $\sum_{i=0}^{\infty} a R^i = a/(1-R)$

In our case $a = 1$

Therfore term $∎_2$ is 1

Therefore equation ∎, i.e., $E(I$ for $I_{odd})$ becomes

$$= \frac{2P1}{[1-(1-P1)(1-P2)]^2} \sum_{i=0}^{\infty} i [(1-P1)(1-P2)]^i [1-(1-P1)(1-P2)]^2$$

$$+$$

$$P1/[1-(1-P1)(1-P2)] \qquad ∎$$

54

Consider the summation to infinity term of equation **B**

This becomes

$$= [1-(1-P1)(1-P2)]^2 \sum_{i=0}^{\infty} i \, [(1-P1)(1-P2)]^i \qquad \textbf{B}_2$$

the term $\textbf{B}_2$ reduces to $(1-P1)(1-P2)$

Therefore equation **B**

$$E(1 \text{ for } 1_{odd}) = \frac{P1(1 + (1 - P1)(1 - P2))}{[1 - (1 - P1)(1 - P2)]^2} \qquad \textbf{C}$$

Expected length for number of iterations being even

For number of iteration being even

$(1) \longrightarrow (2) \longrightarrow (3)$ no. of iterations = 2
$(1) \longrightarrow (2) \longrightarrow (1) \longrightarrow (2) \longrightarrow (3)$ no. of iterations = 4

The probability of number of iterations is even is

$$(1 - P1)^{n/2} \ (1 - P2)^{(n-2)/2} \ P2$$

where $n = 2,4,6,....,n$

Therefore $E(1 \text{ for } 1_{even}) = \displaystyle\sum_{i=0}^{\infty} 2i \ (1-P1)^i \ (1-P2)^{(i-1)} \ P2$

$$= \frac{2P2(1-P1)}{[1-(1-P1)(1-P2)]^2} \ \sum_{i=0}^{\infty} i \ [(1-P1)(1-P2)]^{(i-1)} \ [1-(1-P1)(1-P2)]^2$$

but the summation to infinity term is 1

Therefore $E(1 \text{ for } 1_{even}) = \dfrac{2P2(1-P1)}{[1-(1-P1)(1-P2)]^2}$ **D**

Combining equations **C** and **D** we get

$$E1 = \frac{2P2(1-P1) + P1(1+(1-P1)(1-P2))}{[1-(1-P1)(1-P2)]^2}$$

56

STATISTICAL MODEL FOR ESTIMATION OF
HALSTEAD'S SOFTWARE SCIENCE LENGTH

by

JOJO THOPPIL KUNJUPALU

B.E., University of Madras, 1983
M.S., Kansas State University, 1986

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

There have been many efforts to quantify and predict properties of computer programs. These efforts to quantify properties are generally better aimed at better understanding of the software. Halstead proposed a series of related measures of software complexity that were categorized as software science.

A model (a bipartite graph) can be used to investigate the relationships among Halstead's parameters and other measures. The bipartite digraph is a first step in establishing a theoretical foundation for Halstead's measures. This model will be used to investigate Halstead's measures, other measures, and also actual programs. However, as the first step of our study, the research will be directed towards implementing the model, studying the basic model characteristics, and studying how the model relates to Halstead's length equation.